JaViz: A client/server Java profiling tool

by I. H. Kazi
D. P. Jose
B. Ben-Hamida
C. J. Hescott
C. Kwok
J. A. Konstan
D. J. Lilja
P.-C. Yew

The Java™ programming language, with its portability, object-oriented model, support for multithreading and distributed programming, and garbage collection features, is becoming the language of choice for the development of large-scale distributed applications. Without a suitable performance analysis tool for Java programs, however, it is often difficult to analyze the programs for performance-tuning problems. The profiler included in Sun's Java Development Kit (JDK™) 1.1 does not provide sufficiently detailed trace information to address performance issues in large applications. Also, it does not support the tracing of client/server applications, which are very important for analyzing distributed systems. The JaViz performance analysis tool generates execution traces with sufficient detail to determine program hot spots, including remote method calls, in a distributed Java application program. JaViz provides a graphical display of the program execution tree for the entire distributed application in the form of a call graph for ease of visualization. A number of features allow users to analyze the execution tree for performance-tuning problems more easily than other Java performance monitoring tools. The usability and functionality of the JaViz tool set is demonstrated by applying it to an example distributed Java application program.

The Java** programming language presents several features that appeal to developers of large-scale distributed systems. Features such as "write once, run anywhere"*** portability, portable support for multithreaded programming, support for distributed programming, including remote method invocation, garbage collection, and an appealing object model have encouraged Java use for systems with a size and complexity far beyond small applets. Larger applications, however, encounter performance problems that may never bother users of small applications. The developers of these applications typically encounter four types of problems:

- *Distributed applications.* Large-scale client/server applications distribute objects and then execute across multiple machines. A critical step in performance tuning for these distributed applications is the identification of hot spots where there is extensive remote method invocation (RMI).
- *Inefficient methods.* Methods coded for flexibility and generality can cause significant performance problems when used extensively in larger applications. For example, developers often discover that an inordinate amount of time is spent in certain Java String package methods.
- *Synchronized methods.* Synchronized methods force mutually exclusive access to protected objects. Significant contention due to a synchronized method can cause performance to suffer greatly.

0018-8670/00/$5.00 © 2000 IBM

- *Memory management.* Sun's JDK 1.1 and Java 2 Software Development Kit (SDK), Standard Edition, v 1.2.2 garbage collectors appear to have been optimized for applications with a relatively small memory footprint. As a result, large applications can experience unacceptably large and unpredictable delays during garbage collection.

The JaViz performance visualization tool presented in this paper is currently designed to address primarily the first two of these problems. However, it can be extended to address the remaining two problems as well.

JaViz was developed to supplement, rather than replace, existing Java performance analysis tools such as the profiler included in Sun's JDK 1.1,[1] the HyperProf tool,[2] the ProfileViewer tool,[3] the JProbe** tool,[4] and the OptimizeIt!** tool.[5] Sun's JDK, for instance, provides a profiling option to count method invocations both individually and by caller-callee pairs. It also provides information about average method execution time. The HyperProf tool can display these trace files as a hyperbolic tree, allowing users to look at the time involved in each method, the methods called by each method, and so forth. The ProfileViewer tool also uses the profile generated by Sun's Java virtual machine (Jvm) to display the caller-callee relationships with timing information. Due to the coarse granularity of the underlying trace, however, it is not possible to determine whether methods have high or low execution time variances or whether method execution time varies based on the caller's context and parameters. Also, it is not possible to trace execution threads at all, let alone across Jvm boundaries, when RMI calls are made. The JProbe, the OptimizeIt!, and the Visual Quantify[6] profiling tools provide powerful graphical analyzers to identify performance bottlenecks in Java programs, but they do not currently support coordinated tracing of client/server activities.

IBM's Jinsight[7] is another performance visualization tool for Java application programs. Jinsight uses traces from a modified Java virtual machine (IBM Developer Kit for Windows**, Java Technology Edition, Version 1.1.x and IBM Developer Kit for AIX* [Advanced Interactive Executive], Java Technology Edition, Version 1.1.6) to display performance bottlenecks, object creation, garbage collection, thread interaction, and object references. It also supports a feature to identify and solve memory leaks. Jinsight, however, lacks the capability of tracing client/server activities across multiple Jvms. Thus, while Jinsight is very useful for performance analysis of Java applications running on a single Jvm, it cannot be used to identify performance bottlenecks in distributed Java application programs.

The existing Java profiling tools do not currently support client/server activities, making them of limited use in the analysis of distributed application programs. To extend Java's existing profiling support for a distributed environment, JaViz focuses on a detailed method-level trace with sufficient detail to resolve RMI calls between traces. Given the extremely large amount of data that could be traced, it is particularly important to minimize the tracing overhead both through careful design and by allowing users to control the set of methods to trace. Finally, it is expected that users would be unable to directly perceive and process the large volume of data created by a set of method-level traces. Accordingly, JaViz incorporates visualization and statistical analysis tools to help users understand the trace results.

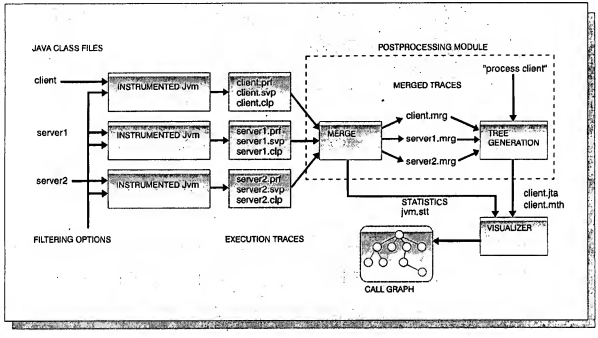
Several functionality, performance, and usability requirements drove the development of this performance visualization tool. Requirements for functionality include:

- The ability to trace and record elapsed time for all method calls executed in a Java virtual machine
- The ability to resolve a method-call trace into an execution tree, with callers being parents of callees. This tree should include all threads of execution, with a thread's run method appearing as the child of the method that invoked it.
- The ability to build a tree of execution that spans multiple Jvms. This tree should be rooted at the call to main from one Jvm, and should include all local and remote method calls executed.
- The ability to correctly build each of these trees from the remaining method calls when certain method calls are omitted for trace-generation efficiency

Performance requires:

- An implementation that does not excessively distort the performance profile of the applications being traced
- An implementation that allows traces to be generated as quickly as feasible
- User control over the methods to be traced. For example, users should be permitted to avoid tracing Java library methods if they are only interested in performance issues in their own code.

Figure 1 The flow of information in the JaViz instrumentation and visualization process



Usability requires:

- An easy-to-use tracing and trace-processing environment
- An easy-to-use interface for exploring trace results

The remainder of this paper describes the JaViz performance analysis tool set and its implementation and use. The next section describes the tools in detail, focusing on what they accomplish and how they work together. The following section describes the implementation of the tools, including the modifications needed to the Jvm to generate traces, the algorithms for resolving RMI calls across traces, and the algorithms for processing traces into execution trees. It is followed by an example of the use of the system. Finally, we present conclusions and describe some future enhancements that are planned for the JaViz tool set.

The JaViz performance visualization tool set

JaViz provides an execution profile of a Java program in the form of an execution tree that shows the caller-callee relationship of the method calls in the program, with the callers being the parents of the called methods in the tree. An execution tree represents the call graph of a single client or server program. Thus, in a client/server environment with n different client or server Jvms, JaViz will generate n different execution trees representing the execution profile of each client or server. The execution tree shows all method calls that were invoked as part of the corresponding client's (or server's) execution. Thus, RMI calls that originated from the client program to be executed on other server Jvms will be included in the client's execution tree. On the other hand, incoming RMI calls that executed on this Jvm are part of another client's execution. Hence, these incoming RMI calls will be excluded from its execution tree.

As shown in Figure 1, JaViz consists of three major components: an *instrumented Jvm* that has been modified to trace method calls, a set of *postprocessing* tools that resolve remote method calls and generate statistics, and a *visualization* tool. The postprocessing and visualization tools are written in the Java language and thus are portable across any platform that supports the language.

Java class files are executed by the instrumented Jvm to generate the execution traces. These trace files are then processed to generate the dynamic execution tree. The first step in the postprocessing phase merges the traces from multiple Jvms to follow the RMI invocations. These merged traces are further processed to build the execution call-graph tree, which can then be displayed by the visualizer. During the merge step, some performance statistics for each method are gathered and fed to the visualizer to be displayed along with each method call. The following subsections provide more details about each of these components of JaViz.

Instrumented Jvm. There are several different techniques for collecting run-time execution information, including techniques such as sampling and direct instrumentation.[8] Sampling requires the running application to be stopped periodically to obtain information on methods that are currently being executed. The accuracy of the information obtained through sampling is determined by the sampling frequency. A higher sampling frequency can provide more detailed information, but this greater detail comes at the expense of greater perturbation of the executing program. Direct instrumentation, on the other hand, adds code to the Jvm to directly measure method execution times. This approach provides more precise information than sampling, since no execution steps are missed. However, the additional instrumentation code may produce greater perturbations than sampling. Nevertheless, JaViz uses direct instrumentation to provide detailed information about each method's execution. To minimize perturbation, the Jvm was modified only to the extent necessary to generate enough trace information to visualize the execution call graph.

As a Java program is executed by the instrumented Jvm, three trace files are generated, as shown in Figure 1. The .prf file contains detailed trace information that records call and return time stamps for every method executed. Invocations of the same method executed under different threads are distinguished from one another by their unique thread identifiers. The other two files record the client/server interaction, if any, that occurs on the Jvm as the program is being executed. The .clp file contains information about all of the outgoing RMI calls from the running Jvm, i.e., identifying information for remote methods invoked by this Jvm. The .svp file records information about all incoming RMI calls, i.e., all of the methods remotely invoked on this Jvm by other Jvms. For the purposes of this explanation, the .clp file is referred to as the *client* profile of remote methods for which the Jvm acts as a client, and the .svp file is referred to as the *server* profile of remote methods for which the Jvm acts as a server. Note that a *server* Jvm may also execute client-type functions and a *client* Jvm may also act as a server to other Jvms.

To reduce the amount of trace data collected, a number of filtering options are available. One option allows the user to eliminate the tracing of Java library calls. This option limits tracing to only the top-level method calls made to any Java library call. The second filtering option specifies a list of classes to be traced. Only method invocations within the objects of those classes and their subclasses are traced. With filtering, the direct caller-callee relationship among the method calls may not be shown in the final call graph if, for example, the caller method is filtered out. The execution tree, however, will have all descendents connected properly to their ancestor node even though some intermediate ancestors may be missing.

Postprocessing. The information in the trace files generated by the instrumented Jvm needs to be further processed to generate the dynamic execution tree of the Java program in a form that facilitates the visualization process. This postprocessing is done in two separate steps. In the first step, called the *merge* step, the client and server profiles are merged to produce complete traces for each Java virtual machine. During this merge process, run-time statistics are gathered for the method calls in the trace files. These statistics are subsequently displayed by the visualizer. The *tree generation* step follows the *merge* step to generate the dynamic execution tree for a given program.

Merge step. Once all of the client and server programs have completed execution, trace files from each Jvm involved in a distributed program are available for further processing. The three trace files generated per Jvm—the detailed trace, the client profile, and the server profile—are merged to produce one detailed trace for each Jvm that contains all of the relevant information, including the client/server activity. Using the client and server profiles of the different Jvms, the merge process tracks remote method calls made by any Jvm from the client-side detailed trace to the server-side detailed trace. At the end of this step, the merged detailed trace of each Jvm contains pointers to the merged trace files of the other Jvms

such that the path of every remote call from the client to the server can be uniquely identified.

Tree generation. The tree generation step analyzes the merged trace files to create an output file containing the dynamic execution tree for a given client or server program. This output file is used by the visualizer to display the call graph. As stated earlier, each .mrg file contains trace information for a particular Jvm with links to other .mrg files for the remote calls made to those Jvms. To generate the dynamic execution tree for a given client or server, the tree generation step must follow these RMI links to the .mrg files of the other Jvms and extract the detailed trace information related to the RMI call to be merged into the execution profile for the desired client or server.

The tree generation step takes the .mrg file corresponding to the desired client or server as its input. It processes each method call in the trace file and writes a corresponding record to the output .jta file. If the current method call being processed is an RMI, it goes to the appropriate .mrg file and picks up the corresponding traces from there. Each record in the .jta file, which corresponds to a single method invocation, contains a pointer to its parent method and a pointer to its last child method (the last method invoked by this method). Each child method has a pointer to its previous sibling method. In addition to the parent-child links to reflect the call graph, each record contains such information as the number of methods invoked by this method, the time when the method started, the time when it completed, the thread executing this method, the method identifier of the method call being represented, and the specific Jvm on which the method is executed. The information provided in the .jta file makes it more efficient for the visualizer to display the call graph. To allow the visualizer to display the corresponding method names, a mapping of method identifiers to method names is provided in a .mth file.

Run-time statistics generation. To facilitate the performance analysis of the call graph, statistical information about each of the methods in the program is gathered in the merge step. Each detailed .prf trace file is analyzed to gather the total number of calls made to each method, the maximum, minimum, and average execution times, and the standard deviation of the execution time for each method. The statistics for all the methods are written in the jvm.stt file. The statistics are subsequently displayed by the visualizer when a node is selected in the call graph.

Visualizer. The *visualizer* is the last component of the JaViz tool set. It reads the .jta output file of the postprocessing step, which contains the dynamic execution tree of a program, and graphically displays it as a tree with detailed node information. The visualizer provides the user with the ability to navigate through the tree nodes, expand and contract the tree, search for methods with particular attributes, and map method attributes to graphical display parameters, such as color, size, and arc width, to make it easier to identify program areas of interest.

The visualizer consists of two windows, the *tree window* and the *node information window*, as shown in Figure 2. The tree window displays the graphic representation of the execution tree. Initially, the tree displays only the root node and its immediate children. The user then can select any child of the root and visualize its children, and so on out to the leaf nodes. At any instance, the tree window shows all of the open nodes and arcs with the currently selected node highlighted by a star pattern. Leaf nodes are displayed with an underscore to indicate that these nodes cannot be explored further.

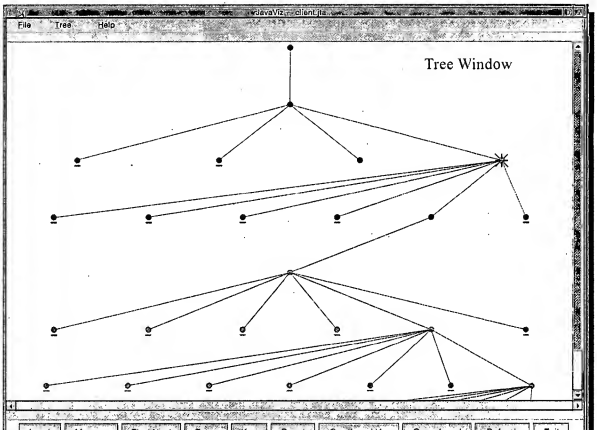
The node information window displays data about the currently selected method node. Table 1 lists the information displayed in this window.

The visualizer provides the user with a collection of functionalities, listed in Table 2, that can be accessed from the tree window through mouse buttons, a menu bar, pop-up menus, and keystrokes. Besides allowing the user to highlight nodes with specific attributes, the mapper also allows the user to specify range queries. Instead of entering all of the different parameters for a query, the user can simply specify the attribute of interest. The visualizer will then map different values (or value ranges) of that attribute to display attributes. For instance, if the user wants to highlight the nodes with different thread identifiers using different node colors, the visualizer will automatically assign a new color for each identifier. Without this feature, the user would have to manually create many individual specific mappings to have the same effect.

Implementation details

This section describes the implementation details of the various components of JaViz. Specifically, the modifications needed to the Jvm to gather the detailed profile traces are described, along with the client/server traces, the algorithms used to generate

Figure 2 The visualizer portion of the JaViz tool set



merged traces by resolving RMI calls across Jvms, and the algorithms used to process traces to build the dynamic execution tree. The implementation of the visualizer is also discussed.

Instrumented Jvm. The profiler that comes with Sun's JDK 1.1.5 provides limited trace information,[9] recording only the cumulative time in milliseconds for each caller-callee pair. Moreover, the profile data do not include any client/server activity. The Jvm used in JaViz is modified to generate more detailed information for each method call as well as to generate traces of client/server programs that span across multiple Jvms. In particular, the Jvm is instrumented to:

- Generate detailed traces of every method call for the program it is executing
- Generate traces to track the client/server interaction for the program in a client/server environment

Table 1 Information displayed in the node information window

Method name	Name of the method represented by the node
Caller name	Name of the method that called the current node
Machine name	Name of the Jvm on which the method executed
Thread number	A number identifying the thread that executed this method instance
This method: Number of calls	The total number of times this method is called in the program
This method: Total time	The total time this method took to execute over all its invocations in the program
This method: Minimum time	The minimum time this method took to execute over all its invocations in the program
This method: Maximum time	The maximum time this method took to execute over all its invocations in the program
This method: Average time	The average execution time of this method over all its invocations in the program
This method: Standard deviation	The standard deviation of the execution times of this method in the program
This call: Call time	The time in microseconds when this method instance started
This call: Return time	The time in microseconds when this method instance completed
This call: Total time	The total execution time, in microseconds, of this method instance

Table 2 Functionalities provided by the visualizer to aid performance analysis

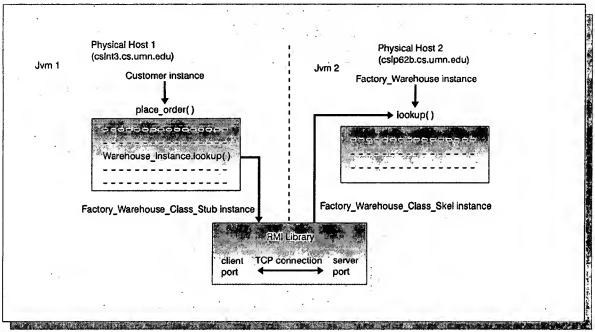
Load	Loads a new trace file and method file for display
Mapper	Maps a specific data query (thread identifier, machine name, method name, and so on) to a particular display style (node or arc color, shape, or size)
Find next	Finds the next node matching the given parameters
Zoom	Zooms in or out, vertically or horizontally
Key	Displays information about the applied mappings
Reset	Resets the root of the displayed tree to the original root
Up one level	Sets the root of the tree to the parent node of the current root (disabled if the root is the original root)
Open level	Displays all the nodes in the next level
Refresh	Redraws the tree
Exit	Exits the visualizer

Detailed trace generation. The trace generation module of the Jvm is modified to record every invocation of a method using time stamps that show the start and end times of the method with microsecond resolution. Additionally, a *thread identifier* is recorded to uniquely identify the thread executing the method. Sun's JDK 1.1.5 Jvm implementation uses a function called java_mon to trace method calls. If the "-prof" profiler flag is set, the java_mon function is called at every method invocation and exit. JaViz uses a similar strategy to trace method calls by invoking its own profiling function instead of the java_mon function. The profiling function creates a new trace record in a buffer for each method entry or exit event. For faster processing, the trace information is stored in main memory and written to external files only when the buffer overflows. Since disk operations are time-consuming, it is important to minimize the number of writes to the external file. Consequently, the trace generation reduces the size of the records stored for the method calls by storing the time values as 32-bit integers. A 64-bit reference time is written on the external file whenever the 32-bit time overflows. Furthermore, the method name associated with each time value is usually quite large, typically hundreds of bytes. Instead of storing the entire name, a 4-byte *method identifier* is used. To generate unique identifiers, a hash table in main memory translates the method names to corresponding identifier values.

Since the number of methods called within a program can be quite large, and since each instance of a method generates a trace entry, the amount of trace data generated for a large application would be enormous. Filtering options are provided with the instrumented Jvm to reduce the amount of trace data collected. When the "exclude Java library calls" filtering option is enabled, the Jvm checks the caller of the current method in the execution stack. If the caller belongs to a *Java* or *Sun* package, which indicates that the caller is a Java library method, the current method is not traced. The second filtering option specifies a list of classes to be traced. For this option, the class hierarchy of the object on which the current method is invoked is checked to see whether it belongs to any of the classes specified. If so, the method is traced. Otherwise, the method invocation is simply ignored.

Client/server trace generation. One of the unique features of JaViz is its ability to track a program's ex-

Figure 3 An example of a remote method invocation in Java

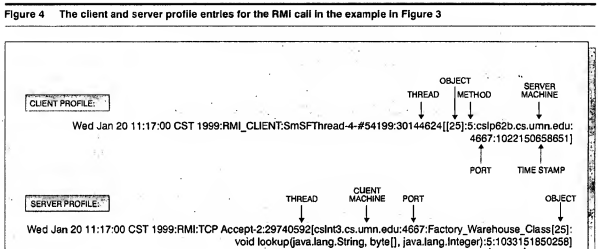


ecution across multiple Jvms. The Java remote method invocation (RMI)[10] facility allows one Jvm to execute a method on another Jvm, which may be executing on a physically distributed processor. In Figure 3, Jvm 1 and Jvm 2 are Java virtual machines running on two different physical hosts. In this particular case, Jvm 2 acts as a server while Jvm 1 acts as a client. The Factory_Warehouse instance is a remote object created on Jvm 2 that is available to Jvm 1. RMI invocation procedures allow the client Customer instance to obtain a reference to the remote Factory_Warehouse instance.

Jvm 1 executes the method place_order within the Customer object. The place_order method invokes Warehouse_Instance.lookup where the Warehouse_Instance object contains the remote reference to the Factory_Warehouse object. Internally, this call is directed to Factory_Warehouse_Class_Stub.lookup, where the Factory_Warehouse_Class_Stub instance is a client-side dummy of the remote object. The *stub* object then establishes a TCP/IP (Transmission Control Protocol/Internet Protocol) connection with the server-side dummy, an instance of Factory_Warehouse_Class_Skel. The stub object on the client side communicates with the *skel* (skeleton) object on the server side through the object serialization protocol for remote invocation.[9] The skel object finally invokes the remote method lookup on the actual remote instance of Factory_Warehouse and returns the result back to the client-side stub method. This then completes the RMI procedure. Note that this RMI procedure is specific to Sun's JDK 1.1.

To trace client/server activities through RMI, every object to be exported to a remote Jvm is given a unique identifier automatically by the server Jvm. Similarly, each method that can be remotely invoked in an exported object is also given a unique (within a class) identifier by the RMI module. For every remote method invoked through RMI, JaViz's modified Jvm records these identifiers at both the client side and the server side. For example, in Figure 3, the Factory_Warehouse instance may be identified as Object 25 on the server. This number will be unique on the server so that the performance analysis tool knows that every remote call to Object 25 is always directed to this Factory_Warehouse instance. Similarly, if the remote method lookup is identified as Method 5, the combination of Object 25 and Method 5 will identify Factory_Warehouse.lookup on the server.

Figure 4 The client and server profile entries for the RMI call in the example in Figure 3



To match corresponding entries in the server and client profiles, the modified Jvm also records the machine (Jvm) names. On the client side, the server machine name on which the call was invoked is recorded. On the server side, the client machine name from which the call originated is also recorded. The client-side port number of the TCP/IP connection used for the remote call is recorded in both the server and the client profiles. The port number is needed to distinguish between remote calls made to the same server from different client Jvms residing on the same physical machine.

The identifier of the thread that invoked the remote call is recorded on the client side in order to map the detailed trace entry of the remote method invocation to the corresponding client profile entry. Similarly, the identifier of the thread where the remote call is received is recorded on the server side. Finally, the time stamps—the time at which the remote call was invoked on the client and the time at which the call was received on the server—are also recorded.

The trace entries in the client and server profiles are shown in Figure 4 for the RMI call Warehouse_Instance.lookup( ) from the client on Jvm 1 to the server on Jvm 2 in the example in Figure 3. The client entry indicates that it is an RMI call to Object 25 for Method 5 on the server running on Machine cslp62b.cs.umn.edu, which in this particular example is the Factory_Warehouse.lookup method on Jvm 2. The trace information also records the identifier of the thread invoking the RMI, which is "30144624," the port number "4667" of the TCP/IP connection, and the time stamp "1022150658651." The corresponding entry in the server profile indicates that it is an incoming RMI call from the client on Machine cslnt3.cs.umn.edu through Port 4667 for Method 5 on Object 25. Thus, the server-side record links back to the entry in the client profile. The entry also records the thread identifier "29740592," and the time stamp, "1033151850258."

Note that the time-stamp values at the client and the server are not the same since the clocks in a distributed system are not guaranteed to be synchronized. JaViz does not depend upon having synchronized clocks on the various client and server machines. This design decision reflects the real-world likelihood that distributed applications, particularly those with shared servers, may be unsynchronized and may indeed be managed separately and therefore hard to synchronize.

The lack of synchronized clocks requires some additional processing and has only minimal consequences. The major additional processing is needed

because only chronological time-stamp order, not time stamps, is used to resolve client/server calls. Indeed, because of variance in network overhead, such times could not be uniquely used in applications with more than one thread or process making RMI calls to the same server. The technique used in JaViz resolves the calls without needing a synchronized clock. A second consequence of unsynchronized clocks is the apparent lack of time continuity when tracing execution across RMI calls. The time displayed in the server might precede the call time, or follow the return time, displayed in the client. This discrepancy can be addressed by rescaling times to create a "plausible global time." Thus far, this discontinuity has not been found to be a significant distraction, particularly since elapsed time is displayed and each call has both call and return time measured on the same clock. A third consequence is a minor limitation on the ability to measure network overhead. It is possible to measure total network overhead by subtracting elapsed time on the server from elapsed time on the client (this measure includes time in the RMI call on both ends). It is not possible, however, to break this down further into call and return time, since the server time cannot be placed within the client window. No user has requested this ability, however, in part because network overhead is predictable, and in part because it is out of the control of the optimizer (except at the granularity of RMI invocations).

Trace generation with the Java 2 environment. JaViz was designed to instrument and profile applications running in the widely used Sun JDK 1.1 Java virtual machine. Java execution environments frequently include just-in-time (JIT) compilers to enhance Java performance.[11] Some changes in the JaViz trace generation module will be needed to retain the accuracy of the tool when used in such execution environments. JaViz already provides the proper hooks into RMI to trace RMI calls, whether the initiator is JIT compiled or bytecode interpreted. Tracing method calls in JaViz requires only that they not be compiled inline. If JIT compilers do compile methods inline, however, we expect declarations to be available to prevent the inline compilation of the particular methods being traced.

Sun's Java 2 SDK, Standard Edition, v 1.2.2 provides extended trace generation support that will allow JaViz to trace JIT-compiled methods. All the detailed tracing information currently captured by the instrumented Jvm of JaViz can also be obtained through the new Java Virtual Machine Profiler Interface (JVMPI)[12] provided in Sun's Java 2 SDK. JVMPI will be able to provide trace information on JIT-compiled methods as well as object instances—two key pieces of information that are missing from the current implementation of JaViz. The only problem is that the resolution will still be milliseconds. As we extend JaViz to the Java 2 SDK, we intend to use JVMPI as much as possible to remove JaViz's dependence on a modified Jvm. The tracing of client/server activity, however, will still need to be done by modifying the RMI library implementation.

Future enhancements to Sun's RMI support will allow IIOP (Internet Inter-Orb [object request broker] Protocol) to be used as a transport protocol, giving interoperability with CORBA** (Common Object Request Broker Architecture**) -based applications and services.[13] Since the Jvm instrumentation of JaViz is restricted to the underlying TCP/IP layer for network connection identification and higher-level object, method, and thread identification, the RMI enhancement to support IIOP is not expected to have any impact on our client/server tracing mechanism.

Postprocessing. The postprocessing step takes the detailed .prf profile of each Jvm, along with the .clp and the .svp files, as input. The merge and tree generation substeps process these input files to produce a dynamic execution tree for the desired client or server. The details of these steps are described in the following subsections.

Merge step. The main part of the merge process is to link client calls recorded in the client profile of a client Jvm to the appropriate entry in the server profile of the remote server Jvm where the method was actually executed. To see how the corresponding client- and server-side entries are matched, consider a simple scenario where there is one client Jvm interacting with one server Jvm. In a single-threaded application, multiple calls made from the client to the same remote object and method on the server can easily be matched using remote object and remote method identifiers recorded in the client- and server-profile entries. The entries with the same remote object and remote method identifiers on both sides can be aligned in chronological order and the entries matched in that order. Note that the clocks on the two machines need not be synchronized for this matching to succeed. Instead, the matching process relies only on the relative order of calls within each Jvm. Since the Jvm is multithreaded, however, there could be a second call made to the server from the client through another thread before the first call even begins, if Java threads are mapped to different

native threads in the Jvm implementation. Thus, the second call may arrive at the server before the first one. The remote object and remote method identifiers are not sufficient to unambiguously link the appropriate calls in this case. Consequently, the port number of the TCP/IP connection is used to resolve this ambiguity.

Consider the following sample trace entries from a client-side Jvm (top) and a server-side Jvm (bottom). The entries are listed in chronological order.

```
port1:myserver:object3:method4
port2:myserver:object3:method4

port2:myclient:object3:method4
port1:myclient:object3:method4
```

In the first two entries, port1 and port2 are the client-side port identifiers of the TCP/IP connection. Relying only on time ordering to match machine name, method identifier, and object identifier, the first entry on the client side would be matched to the first entry on the server side. However, since their port numbers do not match, the first entry on the client side is matched to the second entry on the server side. Because there cannot be more than one connection active at a time through a single TCP/IP port, all RMI calls going through the same TCP/IP connection are guaranteed to be time-ordered. Furthermore, the TCP/IP port numbers active at the same time on the same physical machine for different client Jvm processes are guaranteed to be unique. Even though this situation may be rare in a single-client scenario, it is not unusual when there are multiple clients running on the same physical host interacting with the same server.

In addition to the matching of corresponding client- and server-side entries, the entries in the client profile must be linked to the correct method entries in the detailed profile of the client Jvm. As shown in Figure 3, all client calls originate from an instance of a stub class. All method calls in any stub class are picked up from the detailed profile and matched with entries in the chronologically ordered client profile using the method name, the class name, and the thread identifier. Matching the thread identifiers is necessary since multiple threads could be active at the same time in a Jvm on a multiprocessor machine.

The entries in the server profile also must be linked to the correct method entries in the detailed profile of the server Jvm. Server-side execution of a remote method is initiated by an instance of a skel class (refer to Figure 3) with the invocation of a dispatch method. This dispatch method in turn invokes the desired remote method in the remote object. All dispatch method calls in any skel class are extracted from the detailed profile and matched with entries in the chronologically ordered server profile based on the class name and the thread identifier. Note that the actual method name on the detailed profile entry will always be dispatch.

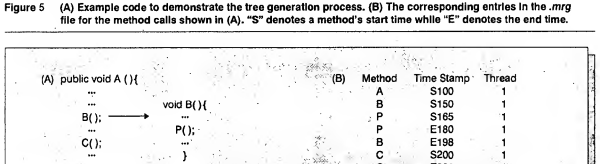
Tree generation. The execution traces in the .mrg files cannot be directly used to display the call graph since the trace data do not provide any explicit parent-child links among the method calls. However, the information included in the traces is sufficient to build the call graph. The main goal of the tree generation step is to make the parent-child relationships among the different methods in a given .mrg file explicit in order to build a dynamic execution tree that can readily be used by the visualizer to display the call graph. The .mrg file lists the start time, end time, and thread identifier of each local method in time order. For remote methods, it also lists the name of the Jvm on which the RMI was executed. The .mrg file for this remote Jvm must be analyzed to build the portion of the execution tree corresponding to the remote method's execution. The input .mrg file may also contain traces of remote methods invoked by other Jvms but executed on the local Jvm. Since these RMI traces are part of a different client program's execution, they must be ignored while building the execution tree. Thus, the tree generation algorithm deals with three main issues:

- Making the parent-child relationship among method calls explicit
- Incorporating any RMI processing performed for the input .mrg file that appears in other .mrg files
- Excluding the traces for remote methods in the input .mrg file that come from other Jvms

These algorithms are now discussed in detail.

Determining parent-child relationships. A stack-based algorithm is used to determine the parent-child relationship among the method calls in the merged traces. Except for one special case involving new thread creation, a method that calls child methods completes its execution only after all of its called methods complete. Since the methods in the merged trace appear ordered by their start times or end times, a method that calls other methods encloses the start and end times of all its children within its

Figure 5 **(A) Example code to demonstrate the tree generation process. (B) The corresponding entries in the *.mrg* file for the method calls shown in (A). "S" denotes a method's start time while "E" denotes the end time.**



own start and end times. Thus, a stack of methods is maintained where a node is pushed onto the stack each time its start time is found and is popped off the stack and written in the output file when its end time is encountered. The order of the methods appearing in the .mrg file ensures that a parent method is not popped off before all of its child methods are pushed onto the stack. This ordering also guarantees that the parent of the most recently inserted method (i.e., the top method) is immediately below the inserted method on the stack.

This single stack of methods works correctly to determine parent-child relationship as long as the program has a single thread of execution. In a multithreaded application, however, the methods of different threads of execution are interleaved in the trace. Thus, it cannot be guaranteed that the parent of the topmost method on the stack is immediately below it. However, with a separate stack for each unique thread of execution, the above algorithm applies to each stack independently. Thus, the complete stack-based algorithm actually uses multiple stacks, one for each newly created thread, to determine the parent-child relationships of the methods.

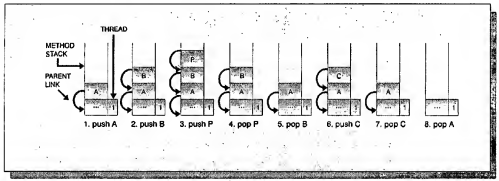
In the example shown in Figure 5(A), Methods B and C are called by Method A and consequently become the children of Method A. Method P is the child of Method B. Assuming that all four methods belong to the same thread (Thread 1), the corresponding .mrg file will have the entries shown in Figure 5(B). Figure 6 shows the application of the stack algorithm to the trace in Figure 5(B). Since the example has only one thread, only a single stack is shown.

The stack-based algorithm assumes that a parent method does not complete until all its children complete. This assumption is valid for all cases except for one special case involving the creation of new threads of execution. In the Java Thread class, two methods are defined that are used for initiating new threads—the start method and the run method. To start a new thread object, the caller invokes the thread object's start method. Start then invokes the thread object's run method, which carries out the actual execution of the new thread. In this situation, the initiating thread is the parent of start while start is the parent of run. However, start merely initiates the run method and often returns before run even begins executing. As a result, the previously described stack-based algorithm cannot determine that start is the parent of run since start will be popped off the stack before run is even pushed on to the stack.

To remedy this problem, the start methods are put into a special queue and linked to their corresponding run methods. When the start time of a start method is encountered, it is inserted in the queue, in addition to the other stack processing. When the start time of a run method is encountered, it is pushed on to its newly created stack, since this is a new thread of execution. Furthermore, instead of the preceding element link, it is linked to the head element of the start queue as its parent. Thus, even if start finishes before its corresponding run and so is popped off the stack, the parent-child relationship is still maintained since the queue holds the parent.

The following is a portion of an execution trace where Method A, in Thread 1, spawns a new thread T (Thread 2), by invoking T.start( ). T.start( ) subse-

Figure 6 Application of the stack-based algorithm for tree generation to the example in Figure 5



quently invokes T.run( ), which creates the new thread.

```
A        S100    1
T.start  S109    1
T.start  E115    1
B        S145    1
B        E167    1
T.run    S204    2
C        S220    2
C        E254    2
T.run    E300    2
...
```

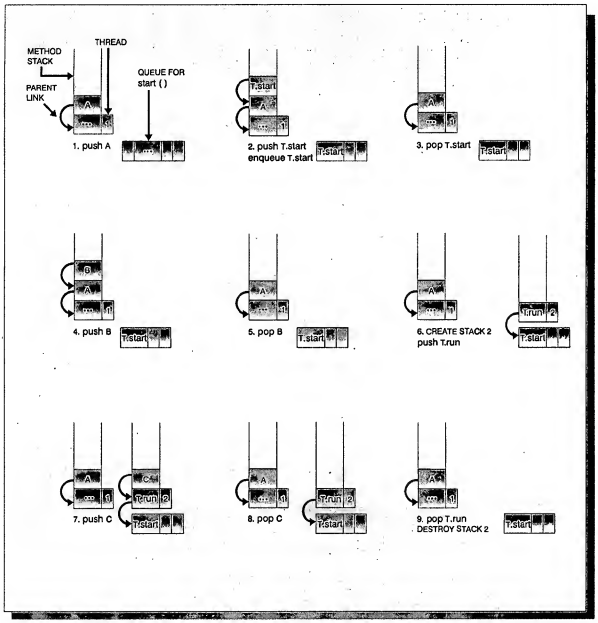
Figure 7 shows how the stack-based tree generation algorithm handles this special case of linking run methods to corresponding start methods when the start returns before run even begins. Since the example deals with two different threads, the processing uses two stacks, one for Thread 1 and the other for Thread 2.

In a Java application program, execution usually begins with the invocation of the main( ) method, which in turn invokes other methods. Thus, the call graph of such an application program would have main as the root of the dynamic execution tree with other methods spawning from main. However, the Java run-time system often invokes a number of methods that are not spawned from main or from any methods within the program itself. Also, in a client/server environment, it is possible to have a method spawned by a client in the server process. A tree rooted at main will not include such methods since they are not spawned by main. The concept of the *virtual root*, which is the root node of any dynamic execution tree, is introduced to handle such cases. This node ties together main and any other methods that are spawned by the run-time system or by a different client or server.

RMI processing. For RMI calls invoked by the input client program, the merged trace for a particular Jvm contains an entry with a link to the remote Jvm that executed the RMI. This entry, which is identified by a stub method call, contains the server machine name, the remote object name, the remote method name, the identifier of the thread that executed the remote method, and the RMI time stamp. The server machine name can be used to identify the corresponding .mrg file that needs to be processed. Within the remote trace file, the appropriate RMI entry is identified by matching the remote object name, the remote method name, the thread identifier, and the time-stamp values obtained earlier from the client profile. The RMI traces are collected from the remote file until the end time stamp of the remote method is encountered. However, any methods that started before the RMI began and finished before the RMI returned are not included since these are part of the profile of the server executing the RMI call.

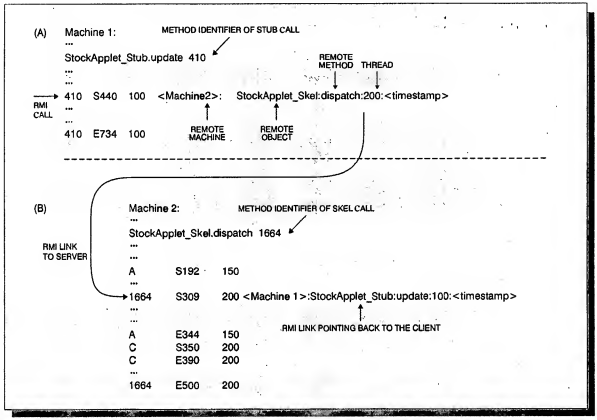
Figure 8 shows the .mrg file entries for an RMI call from a client on Machine 1 to a server on Machine 2. The RMI call on the client machine, which is identified by the StockApplet_Stub.update call, is deter-

Figure 7 Application of the stack-based algorithm to handle the *start* and *run* methods used to initiate a new thread



mined to be served by the StockApplet_Skel.dispatch method on Machine 2. Thus, the tree generation process jumps to the .mrg file of Machine 2 and finds the appropriate RMI entry, which is identified by the StockApplet_Skel.dispatch call and by matching the thread identifier and the time stamp. Once in this file, it processes all entries until the end time stamp of StockApplet_Skel.dispatch (Method 1664) is found. However, Method A on Machine 2 started before Method 1664 and, hence, cannot be part of the RMI. Thus, when the RMI traces are processed, the entry corresponding to Method A is ignored.

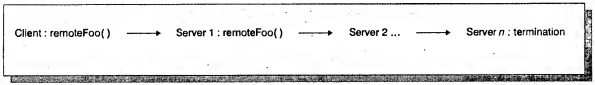
Figure 8 **(A) The *.mrg* file entries for a client running on Machine 1; (B) the *.mrg* file entries for a server running on Machine 2**



Excluding traces of other programs. In the input client trace file, there may be entries for RMI traces, identified by skel method entries that were remotely executed on this Jvm. Since these traces are part of an RMI call for another client, they belong to the execution tree of that other client program. Hence, they should be excluded when building the execution tree for the current program. However, there may be method calls that started after the execution of the RMI but did not end before the RMI returned. These methods are part of the current profile and so should be included. Thus, when a skel method call is encountered, the method calls that started before, but finished within its boundary, are included in the call graph. The other method calls within the boundary of the skel call are excluded. In the example shown in Figure 8, if Machine 2 is the client being processed, then the entries corresponding to the RMI call within the boundary of Method 1664 will be ignored. However, Method A, which started before the RMI but completed within its boundary, must be included as it is part of the input client.

Run-time statistics generation. Each detailed .prf trace file is analyzed to generate statistical information for each method. The statistical data are stored in a hash table keyed by method identifiers for faster access. The statistics generation step reads the entries in the .prf file and updates the hash table entry for the current method. The number-of-times-called value is incremented each time a method entry is encountered. The maximum and minimum execution time values are updated with the current execution time values, if needed. For the average and standard deviation calculation, the appropriate execution time values are accumulated. Once all of the entries are processed, the average and the standard deviation of the execution time for each method are calculated and stored

Figure 9 Propagation of RMI calls in the example program



in the hash table. Finally, the statistics are copied from the hash table to the jvm.stt file (see Figure 1).

Visualizer. The visualizer takes the .jta file and the corresponding .mth file as its inputs. Since it processes the execution tree in the .jta file in a top-down manner, it needs to retrieve the root node first. The root node, which is the virtual root that ties together all of the first level calls, is stored at the end of the .jta file since it is the last method to finish. Thus, the visualizer opens the .jta file and jumps directly to the last record in the file to retrieve the root node information. Each node in the .jta file contains pointers to its last child and its previous sibling, and a count of the number of children it has. After reading this root node, the visualizer knows the location of the last child of the root and the total number of children it has. The visualizer then initiates a loop to jump from child node to child node following the previous-sibling links. While traversing the nodes, it gathers all the pertinent information for each node. Once all the children of the root are traversed, they are displayed.

The visualizer initially displays only the first level of the tree since the .jta file may contain a large number of nodes. Traversing all of them while keeping each node's information in memory could be quite time consuming and may overload the memory. Once the root and its children are displayed, the visualizer allows the user to expand and explore the tree by clicking on the displayed nodes or by using a keyboard navigation technique developed for this purpose.

To minimize the I/O overhead between the visualizer and the .jta and .mth files, flags are added to the nodes to differentiate, for instance, between a node that is not loaded in memory and a node that is already loaded but not yet displayed. In the latter case, the flag will indicate that an I/O to the .jta file is not necessary if the user clicks on its parent to open the node. This feature thus saves processing time.

The visualizer's mapper function offers the user the ability to manually construct queries through a control panel. The visualizer maintains a list of queries, to be processed, in memory. When the user enters a query through the control panel, it is added to this query list. When a query is applied, the visualizer scans the list of queries to check whether a query matches the node's data. If a match is found, the node is displayed with the parameters specified by the query. The process is repeated for each node until all matching nodes are displayed.

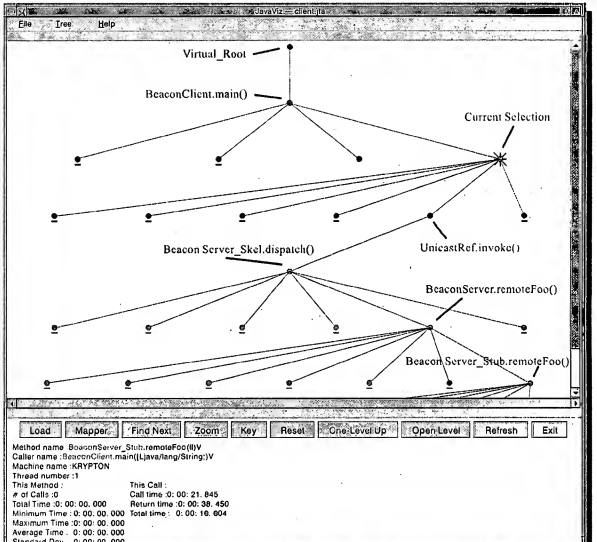
The visualizer also provides a "*find-next*" capability to allow the user to search through the dynamic execution tree for a specific node matching some given characteristics, using either a depth-first or a breadth-first search to locate and select the nodes. This function moves from node to node comparing the node's data with the given query and stopping when it locates a node that matches the user's request. It resumes when the user again clicks the "find-next" button. The find-next function can start from either the current node or from the root node. When the current node is selected as the starting point, only the subtree below the node is searched. Otherwise, the entire tree is searched.

Example

An RMI-based example Java client/server program is used to demonstrate how JaViz works. The program uses multiple servers with each server invoking a remote call to the next server. As shown in Figure 9, a client initially invokes a remote method (remoteFoo) on Server 1. Server 1 in turn invokes the remote method on Server 2. This process continues until the remote call reaches a termination server. While this example is overly simplified, it clearly demonstrates the possibilities of more complex visualizations.

The example Java code consists of two classes—BeaconClient and BeaconServer. BeaconServer contains

Figure 10 Portion of the client's call graph as displayed by the visualizer of JaViz



the code necessary to make the RMI bindings, as well as the code needed to start the daemon thread. It also contains the remote method remoteFoo. When the server is started, two parameters must be given—the name of the current server and the name of the server to which the remote call should be forwarded. If "end" is specified as the second parameter, the server is marked as a termination server to ensure that no remote method calls are made from that server. A typical setup might include the following steps. (It is assumed that the path and class path are set properly and that the platform used is Microsoft Windows NT**.)

1. Start rmiregistry_g
2. Start java_g -um:s1 BeaconServer s1 s2
3. Start java_g -um:s2 BeaconServer s2 s3
4. Start java_g -um:s3 BeaconServer s3 end
5. java_g -um:client1 BeaconClient s1

Figure 11 Data on the *UnicastRef.invoke( )* method call in the node information window of the visualizer

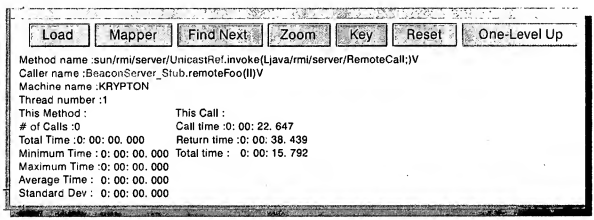
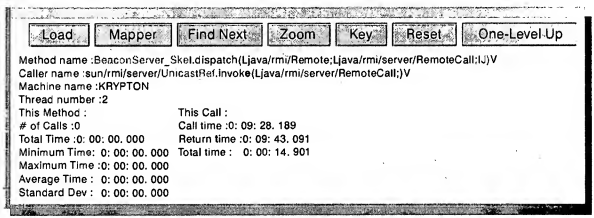


Figure 12 Data on the *BeaconServer_Skel.dispatch( )* method call in the node information window of the visualizer

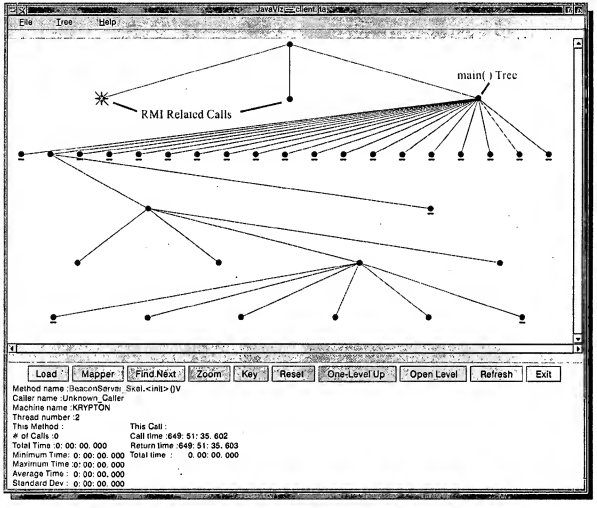


Step 1 simply starts the RMI registry service. Step 2 creates a BeaconServer instance named *s1* that forwards all calls to the BeaconServer instance named *s2*. Server *s2* is set to forward all calls to BeaconServer instance *s3* in Step 3. Step 4 creates *s3* and marks it as a termination server. Finally, Step 5 executes the BeaconClient instance to start the remote call on *s1*. Note that the modified JaViz Jvm and RMI registry are used to execute the Java code. The -um option on the Java command line enables the trace generation by the instrumented Jvm.

After the completion of Step 5, four .prf files are generated, one for the client and one each for the three servers. These .prf files are then merged to create the corresponding .mrg files. To visualize the call-graph profile of the client process client1, the execution tree must be generated by applying the tree generation step on the client's merged trace file client1.mrg. The tree generation step creates the client1.jta file, which contains the execution tree representation of the client process. Once the tree is generated, it is ready for visualization.

Loading the client1.jta file into the visualizer displays the call graph for the client. Figure 10 shows the top portion of the call graph as displayed by the visualizer. The root of the tree, as discussed earlier, is

Figure 13 Portion of the call graph for Server 1



a virtual node that is used to tie together the roots of all of the call trees generated by a particular Jvm. Below the virtual root is the node representing the method main( ). This node has four nodes stemming from it representing four method calls invoked by main. The first two deal with RMI security. The third calls the Naming.lookup method to find the server and initialize the appropriate stub.

The fourth method invoked by main( ) is the remoteFoo( ) method, which is the currently selected node in Figure 10. The method name field in the node information window indicates that this method is invoked on the BeaconServer_Stub object. The BeaconServer_Stub object itself invokes several other methods that set up the RMI connection to the server. The fifth method called by remoteFoo( ) is the Unicast.invoke( ) method; its node information is shown in Figure 11. This method creates the actual invocation on the remote machine. Following the call graph down leads to the child node of Unicast.invoke named BeaconServer_Skel.dispatch( ), which is the node corresponding to the method call in Figure 12. This node contains a different thread number than its parent since the method was invoked on a separate Jvm running a different thread. This difference can be seen by applying the "Jvm ID" option in the visualizer's mapper, which results in method calls on

different Jvms to be displayed in different colors, as shown in Figure 10.

The dispatch method calls four methods to obtain the parameters that are passed to it. It then invokes the BeaconServer.remoteFoo method located on the current server. The remoteFoo method eventually calls the BeaconServer_Stub.remoteFoo, which is the remoteFoo method of the next server. This process is repeated two more times until it eventually reaches the last (i.e., termination) server. A total of four Jvms are used, one for the client and one for each of the three servers. These four unique Jvms can be distinguished by the four different node colors used to represent them in the full call graph (not shown here).

To visualize how the RMI calls are handled on the server side, we need to apply the tree generation step on the merged trace file of the particular server (e.g., s1.mrg) and then load the corresponding .jta file in the visualizer. In the example used, each server merely executes the RMI call that was initiated by the client and then forwarded to each successive server. Thus, most of each server's execution becomes part of the client profile. The call graph for a server itself contains two method calls corresponding to the RMI in addition to the execution tree under its main( ) method.

Viewing the first server's execution tree (corresponding to the file s1.jta) shows the calls handled by the server for the client's RMI call. The server call graph is shown in Figure 13. The first two nodes correspond to the client's RMI call. Notice that each contains its own separate thread number, which is shown as a different color in Figure 13. The third node is the main( ) method executed when the server is first started. The RMI call was invoked by the client on the server and, hence, the nodes related to the RMI call are shown as the children of the virtual root and not as part of main( )'s tree. The ordering of the called methods (from left to right) under a caller in the call graph is determined by the time the callee finishes. Even though main( ) started before the methods related to the RMI began, the RMI calls finished before main( ). Thus, the RMI nodes appear before main( ) in the call graph.

Conclusion

The JaViz performance analysis tool has been developed primarily to address the performance tuning problems encountered when developing large-scale distributed Java application programs. In addition to generating traces for each individual method instance in a program, JaViz has the unique ability to trace multiple threads of execution and method calls (i.e., RMIs) that span multiple Jvms. Besides identifying program hot spots, the "per instance" traces allow program developers to determine whether method execution times have high or low variances and whether method execution times vary with the caller's context and parameters. The traces for different threads provide a means for analyzing multithreaded applications, while RMI traces make it possible to identify hot spots in client/server-based applications. Thus, JaViz appears to be very useful as a performance analysis tool for the development of large-scale client/server-based Java applications, such as those built on the IBM SanFrancisco* framework.[14] JaViz provides an easy-to-use interface with some useful capabilities, such as applying specific queries and finding method calls with specific parameters, to analyze the execution tree. It also provides the user with the ability to control what method calls are to be traced through a filtering interface.

We are currently enhancing the features available in JaViz to make it more powerful. To provide users with more control over what to trace, we are designing other filtering options, such as specifying what not to trace (as well as what to trace) and specifying wild-card options, such as "trace all com.ibm.sf.gf.* method calls." With more filtering options available, we can further reduce the size of the trace data generated. This reduction improves the processing time of later steps in the JaViz tool set to reduce the performance perturbation introduced by JaViz. We also plan to add a visual interface to JaViz that will allow the users to specify tracing options more easily.

To speed up the trace generation process, we will compress the trace file size by writing the trace outputs in a binary format. Currently, the traces are converted to text format before being written to the output file. This text conversion requires a substantial amount of CPU time, which causes perturbations to the execution of the application program. Using a binary format for the traces will reduce the amount of perturbation. While we do not have detailed information about the perturbation overhead of JaViz, nor the time required to generate and visualize a trace, we do have some anecdotal experience. In one instance, for example, we found that a trace file of 17 megabytes required approximately 30 minutes to process and display on a Pentium** II processor, with Windows NT, Version 4.0, running at 400 megahertz.

Additional information will be incorporated into the visualizer window to enhance the visualization process. For example, a feature that would be useful in a large tree is to pop up the call graph for a selected node so that the user can readily visualize the path taken from the root to reach the current method call. This feature is planned for the next version of JaViz. With further enhancements, we hope to broaden the utility of JaViz as a performance analysis tool for large-scale Java application program development.

A preliminary version of the JaViz performance analysis tool can be downloaded for experimental purposes.[15] Using this version, the largest program that we have attempted to trace consisted of approximately 130 unique methods with 521410 unique instances of these methods. This program is a client/server application for stock updates. There is a single server and one or more clients. Each client application requests the server for a stock update notification. The server sends the update information back to the requesting client for display. The execution time for this program increased by a factor of approximately 1.5 due to the overhead of the tracing process. The merging and postprocessing steps required about 30 minutes before the final trace could be visualized.

Acknowledgment

The SanFrancisco Performance Team at IBM Rochester has been very helpful in providing feedback for improving the performance and usability of the JaViz tool. Their comments and suggestions are greatly appreciated.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Microsoft Corporation, KL Group, Inc., Intuitive Systems, Inc., the Object Management Group, or Intel Corporation.

Cited references and note

1. Java Development Kit Version 1.1, http://java.sun.com/products/jdk/1.1.
2. HyperProf(v.1.3)—Java Profile Browser, http://www.physics.orst.edu/~bulatov/HyperProf.
3. ProfileViewer, http://www.inetmi.com/~gwhi/ProfileViewer/ProfileViewer.html.
4. JProbe Profiler, http://www.in-gmbh.de/english/tools/java/jprobe.htm.
5. OptimizeIt! The Ultimate Java Performance Profiler, http://www.optimizeit.com.
6. Visual Quantify, http://www.sys-con.com/java/reviews/quantify/index.html.
7. Jinsight, http://www.alphaworks.ibm.com/formula/jinsight.
8. R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons Inc., New York (1991).
9. This also applies to JDK 1.1.7.
10. Remote Method Invocation Specification, http://java.sun.com/products/jdk/1.1/docs/guide/rmi/spec/rmiTOC.doc.html.
11. I. H. Kazi, H. Chen, B. Stanley, and D. J. Lilja, "Techniques for Obtaining High Performance in Java Programs," to be published in *ACM Computing Surveys*.
12. Java Virtual Machine Profiler Interface (JVMPI), http://www.javasoft.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html.
13. Java-Based Distributed Computing: RMI and IIOP in Java, http://www.javasoft.com/pr/1997/june/statement970626-01.html.
14. R. Christ, S. L. Halter, K. Lynne, S. Meizer, S. J. Munroe, and M. Pasch, "SanFrancisco Performance: A Case Study in Performance of Large-Scale Java Applications," *IBM Systems Journal* **39**, No. 1, 4–20 (2000, this issue).
15. JaViz: A Client/Server Java Profiling Tool, http://www.cs.umn.edu/Research/JaViz.

Accepted for publication September 28, 1999.

Iffat H. Kazi *Department of Electrical and Computer Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, Minnesota 55455 (electronic mail: ihkazi@ece.umn.edu).* Ms. Kazi is a Ph.D. candidate in electrical engineering at the University of Minnesota, where she received an M.S. degree, also in electrical engineering, in 1998. She received a B.Sc. degree in computer science and engineering in 1994 from the Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, and later was a lecturer in its Department of Computer Science and Engineering. Her main research interests include parallel processing, dynamic program optimization, and high-performance computer architecture. She is a student member of the IEEE Computer Society.

Davis P. Jose *Department of Computer Science and Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, Minnesota 55455 (electronic mail: jose@cs.umn.edu).*

Badis Ben-Hamida *Inxight Software, Inc., 3400 Hillview Avenue, Palo Alto, California 94304.* Mr. Ben-Hamida received an M.S. degree in computer science from the University of Minnesota in 1998.

Christian J. Hescott *Department of Electrical and Computer Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, Minnesota 55455 (electronic mail: hesco001@ece.umn.edu).* Mr. Hescott received a B.S. degree in computer engineering from the University of Minnesota in 1999. He is currently pursuing an M.S. degree in computer engineering at the University of Minnesota. His research interests include dynamic and adaptive reconfigurable hardware as well as bio-inspired solutions for computer architecture.

Chris Kwok *6429 City West Parkway, Eden Prairie, Minnesota 55344.* Mr. Kwok graduated from the University of Minnesota in June 1999 with an M.S. degree in computer and information sciences. His areas of interest include object-oriented programming, Java development, and e-commerce.

Joseph A. Konstan *Department of Computer Science and Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, Minnesota 55455 (electronic mail: konstan@cs.umn.edu).* Dr. Konstan is associate professor of computer science and engineering at the University of Minnesota. Since earning the Ph.D. degree in user interface toolkit technology from the University of California, Berkeley, in 1993, Dr. Konstan has worked on a variety of human-computer interaction projects focused on visualization, multimedia, and information filtering. He is an ACM lecturer and currently serves as editor of the ACM *SIGCHI Bulletin*.

David J. Lilja *Department of Electrical and Computer Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, Minnesota 55455 (electronic mail: lilja@ece.umn.edu).* Dr. Lilja received Ph.D. and M.S. degrees in electrical engineering from the University of Illinois at Urbana-Champaign and a B.S. degree in computer engineering from Iowa State University in Ames. He is currently an associate professor in the Department of Electrical and Computer Engineering and a Fellow of the Minnesota Supercomputing Institute at the University of Minnesota. He also is a member of the graduate faculty in the program in computer science and the program in scientific computation, and was the founding director of graduate studies for the program in computer engineering. He has served on the program committees of numerous conferences, is an associate editor for the *IEEE Transactions on Computers*, and is a distinguished visitor of the IEEE Computer Society. His main research interests include high-performance computer architecture, parallel processing, and computer systems performance analysis, with a special emphasis on the interaction of software and compilers with the architecture. He is a senior member of the IEEE Computer Society, a member of the ACM, and is a registered professional engineer.

Pen-Chung Yew *Department of Computer Science and Engineering, University of Minnesota, 200 Union Street SE, Minneapolis, Minnesota 55455 (electronic mail: yew@cs.umn.edu).* Dr. Yew has been a professor in the Department of Computer Science and Engineering at the University of Minnesota since 1994. Previously, he was an associate director of the Center for Supercomputing Research and Development at the University of Illinois. He is an IEEE Fellow. His research interests include computer architecture, high-performance multiprocessor system design, and performance evaluation.

Reprint Order No. G321-5718.